The multiplication `X*X` waits until `X` is bound. The first `Browse` immediately displays `start`. The second `Browse` waits for the multiplication, so it displays nothing yet. The `{Delay 10000}` call pauses for 10000 milliseconds (i.e., 10 seconds). `X` is bound only after the delay continues. When `X` is bound, then the multiplication continues and the second browse displays 9801. The two operations `X=99` and `X*X` can be done in any order with any kind of delay; dataflow execution will always give the same result. The only effect a delay can have is to slow things down. For example:

```
declare X in
thread {Browse start} {Browse X*X} end
{Delay 10000} X=99
```

This behaves exactly as before: the browser displays 9801 after 10 seconds. This illustrates two nice properties of dataflow. First, calculations work correctly independent of how they are partitioned between threads. Second, calculations are patient: they do not signal errors, but simply wait.

Adding threads and delays to a program can radically change a program's appearance. But as long as the same operations are invoked with the same arguments, it does not change the program's results at all. This is the key property of dataflow concurrency. This is why dataflow concurrency gives most of the advantages of concurrency without the complexities that are usually associated with it.

1.12 State

How can we let a function learn from its past? That is, we would like the function to have some kind of internal memory, which helps it do its job. Memory is needed for functions that can change their behavior and learn from their past. This kind of memory is called *explicit state*. Just like for concurrency, explicit state models an essential aspect of how the real world works. We would like to be able to do this in the system as well. Later in the book we will see deeper reasons for having explicit state. For now, let us just see how it works.

For example, we would like to see how often the `FastPascal` function is used. Is there some way `FastPascal` can remember how many times it was called? We can do this by adding explicit state.

A memory cell

There are lots of ways to define explicit state. The simplest way is to define a single *memory cell*. This is a kind of box in which you can put any content. Many programming languages call this a "variable". We call it a "cell" to avoid confusion with the variables we used before, which are more like mathematical variables, i.e., just short-cuts for values. There are three functions on cells: `NewCell` creates a new cell, `:=` (assignment) puts a new value in a cell, and `@`

(access) gets the current value stored in the cell. Access and assignment are also called read and write. For example:

```
declare
C={NewCell 0}
C:=@C+1
{Browse @C}
```

This creates a cell `C` with initial content 0, adds one to the content, and then displays it.

Adding memory to `FastPascal`

With a memory cell, we can let `FastPascal` count how many times it is called. First we create a cell outside of `FastPascal`. Then, inside of `FastPascal`, we add one to the cell's content. This gives the following:

```
declare
C={NewCell 0}
fun {FastPascal N}
   C:=@C+1
   {GenericPascal Add N}
end
```

(To keep it short, this definition uses `GenericPascal`.)

1.13 Objects

Functions with internal memory are usually called *objects*. The extended version of `FastPascal` we defined in the previous section is an object. It turns out that objects are very useful beasts. Let us give another example. We will define a counter object. The counter has a cell that keeps track of the current count. The counter has two operations, `Bump` and `Read`. `Bump` adds one and then returns the resulting count. `Read` just returns the count. Here is the definition:

```
declare
local C in
   C={NewCell 0}
   fun {Bump}
      C:=@C+1
      @C
   end
   fun {Read}
      @C
   end
end
```

There is something special going on here: the cell is referenced by a local variable, so it is completely invisible from the outside. This property is called *encapsu-*

lation. It means that nobody can mess with the counter's internals. We can guarantee that the counter will always work correctly no matter how it is used. This was not true for the extended `FastPascal` because anyone could look at and modify the cell.

We can bump the counter up:

```
{Browse {Bump}}
{Browse {Bump}}
```

What does this display? `Bump` can be used anywhere in a program to count how many times something happens. For example, `FastPascal` could use `Bump`:

```
declare
fun {FastPascal N}
   {Browse {Bump}}
   {GenericPascal Add N}
end
```

1.14 Classes

The last section defined one counter object. What do we do if we need more than one counter? It would be nice to have a "factory" that can make as many counters as we need. Such a factory is called a *class*. Here is one way to define it:

```
declare
fun {NewCounter}
C Bump Read in
   C={NewCell 0}
   fun {Bump}
      C:=@C+1
      @C
   end
   fun {Read}
      @C
   end
   counter(bump:Bump read:Read)
end
```

`NewCounter` is a function that creates a new cell and returns new `Bump` and `Read` functions for it. Returning functions as results of functions is another form of higher-order programming.

We group the `Bump` and `Read` functions together into one compound data structure called a *record*. The record `counter(bump:Bump read:Read)` is characterized by its *label* `counter` and by its two *fields*, called `bump` and `read`. Let us create two counters:

```
declare
Ctr1={NewCounter}
```

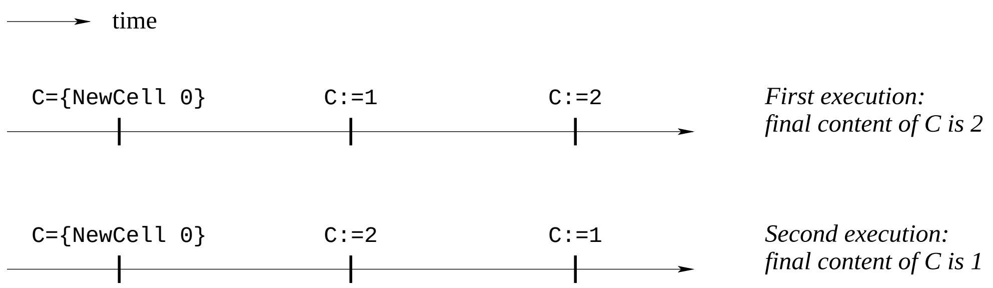


Figure 1.4: All possible executions of the first nondeterministic example

```
Ctr2={NewCounter}
```

Each counter has its own internal memory and its own `Bump` and `Read` functions. We can access these functions by using the "`.`" (dot) operator. `Ctr1.bump` accesses the `Bump` function of the first counter. Let us bump the first counter and display its result:

```
{Browse {Ctr1.bump}}
```

Towards object-oriented programming

We have given an example of a simple class, `NewCounter`, that defines two operations, `Bump` and `Read`. Operations defined inside classes are usually called *methods*. The class can be used to make as many counter objects as we need. All these objects share the same methods, but each has its own separate internal memory. Programming with classes and objects is called *object-based programming*.

Adding one new idea, *inheritance*, to object-based programming gives *object-oriented programming*. Inheritance means that a new class can be defined in terms of existing classes by specifying just how the new class is different. We say the new class inherits from the existing classes. Inheritance is a powerful concept for structuring programs. It lets a class be defined incrementally, in different parts of the program. Inheritance is quite a tricky concept to use correctly. To make inheritance easy to use, object-oriented languages add special syntax for it. Chapter 7 covers object-oriented programming and shows how to program with inheritance.

1.15 Nondeterminism and time

We have seen how to add concurrency and state to a program separately. What happens when a program has both? It turns out that having both at the same time is a tricky business, because the same program can give different results from one execution to the next. This is because the order in which threads access the state can change from one execution to the next. This variability is called *nondeterminism*. Nondeterminism exists because we lack knowledge of the exact time when each basic operation executes. If we would know the exact time, then there would be no nondeterminism. But we cannot know this time, simply because threads are *independent*. Since they know nothing of each other, they also do not know which instructions each has executed.

Nondeterminism by itself is not a problem; we already have it with concurrency. The difficulties occur if the nondeterminism shows up in the program, i.e., if it is *observable*. (An observable nondeterminism is sometimes called a *race condition*.) Here is an example:

```
declare
C={NewCell 0}
thread
   C:=1
end
thread
   C:=2
end
```

What is the content of `C` after this program executes? Figure 1.4 shows the two possible executions of this program. Depending on which one is done, the final cell content can be either 1 or 2. The problem is that we cannot say which. This is a simple case of observable nondeterminism. Things can get much trickier. For example, let us use a cell to hold a counter that can be incremented by several threads:

```
declare
C={NewCell 0}
thread I in
   I=@C
   C:=I+1
end
thread J in
   J=@C
   C:=J+1
end
```

What is the content of `C` after this program executes? It looks like each thread just adds 1 to the content, making it 2. But there is a surprise lurking: the final content can also be 1! How is this possible? Try to figure out why before continuing.

Interleaving

The content can be 1 because thread execution is *interleaved*. That is, threads take turns each executing a little. We have to assume that any possible interleaving can occur. For example, consider the execution of Figure 1.5. Both `I` and

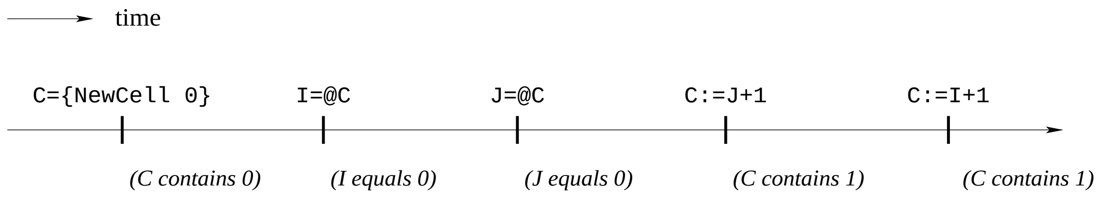


Figure 1.5: One possible execution of the second nondeterministic example

`J` are bound to 0. Then, since `I+1` and `J+1` are both 1, the cell gets assigned 1 twice. The final result is that the cell content is 1.

This is a simple example. More complicated programs have many more possible interleavings. Programming with concurrency and state together is largely a question of mastering the interleavings. In the history of computer technology, many famous and dangerous bugs were due to designers not realizing how difficult this really is. The Therac-25 radiation therapy machine is an infamous example. It sometimes gave its patients radiation doses that were thousands of times greater than normal, resulting in death or serious injury [112].

This leads us to a first lesson for programming with state and concurrency: if at all possible, do not use them together! It turns out that we often do not need both together. When a program does need to have both, it can almost always be designed so that their interaction is limited to a very small part of the program.

1.16 Atomicity

Let us think some more about how to program with concurrency and state. One way to make it easier is to use atomic operations. An operation is *atomic* if no intermediate states can be observed. It seems to jump directly from the initial state to the result state.

With atomic operations we can solve the interleaving problem of the cell counter. The idea is to make sure that each thread body is atomic. To do this, we need a way to build atomic operations. We introduce a new language entity, called *lock*, for this. A lock has an inside and an outside. The programmer defines the instructions that are inside. A lock has the property that only one thread at a time can be executing inside. If a second thread tries to get in, then it will wait until the first gets out. Therefore what happens inside the lock is atomic.

We need two operations on locks. First, we create a new lock by calling the function `NewLock`. Second, we define the lock's inside with the instruction **`lock`** `L` **`then ... end`**, where `L` is a lock. Now we can fix the cell counter:

```
declare
C={NewCell 0}
L={NewLock}
thread
   lock L then I in
```

```
         I=@C
         C:=I+1
      end
   end
   thread
      lock L then J in
         J=@C
         C:=J+1
      end
   end
```

In this version, the final result is always 2. Both thread bodies have to be guarded by the same lock, otherwise the undesirable interleaving can still occur. Do you see why?

1.17 Where do we go from here

This chapter has given a quick overview of many of the most important concepts in programming. The intuitions given here will serve you well in the chapters to come, when we define in a precise way the concepts and the computation models they are part of.

1.18 Exercises

1. Section 1.1 uses the system as a calculator. Let us explore the possibilities:

   (a) Calculate the exact value of 2^{100} *without* using any new functions. Try to think of short-cuts to do it without having to type `2*2*2*...*2` with one hundred 2's. Hint: use variables to store intermediate results.

   (b) Calculate the exact value of 100! without using any new functions. Are there any possible short-cuts in this case?

2. Section 1.3 defines the function `Comb` to calculate combinations. This function is not very efficient because it might require calculating very large factorials. The purpose of this exercise is to write a more efficient version of `Comb`.

   (a) As a first step, use the following alternative definition to write a more efficient function:

   $$\begin{pmatrix} n \\ r \end{pmatrix} = \frac{n \times (n-1) \times \cdots \times (n-r+1)}{r \times (r-1) \times \cdots \times 1}$$

   Calculate the numerator and denominator separately and then divide them. Make sure that the result is 1 when $r = 0$.

(b) As a second step, use the following identity:

$$\binom{n}{r} = \binom{n}{n-r}$$

to increase efficiency even more. That is, if $r > n/2$ then do the calculation with $n - r$ instead of with r.

3. Section 1.6 explains the basic ideas of program correctness and applies them to show that the factorial function defined in Section 1.3 is correct. In this exercise, apply the same ideas to the function `Pascal` of Section 1.5 to show that it is correct.

4. What does Section 1.7 say about programs whose time complexity is a *high-order* polynomial? Are they practical or not? What do you think?

5. Section 1.8 defines the lazy function `Ints` that lazily calculates an infinite list of integers. Let us define a function that calculates the sum of a list of integers:

```
fun {SumList L}
   case L of X|L1 then X+{SumList L1}
   else 0 end
end
```

What happens if we call `{SumList {Ints 0}}`? Is this a good idea?

6. Section 1.9 explains how to use higher-order programming to calculate variations on Pascal's triangle. The purpose of this exercise is to explore these variations.

   (a) Calculate individual rows using subtraction, multiplication, and other operations. Why does using multiplication give a triangle with all zeroes? Try the following kind of multiplication instead:

   ```
   fun {Mul1 X Y} (X+1)*(Y+1) end
   ```

   What does the 10th row look like when calculated with `Mul1`?

   (b) The following loop instruction will calculate and display 10 rows at a time:

   ```
   for I in 1..10 do {Browse {GenericPascal Op I}} end
   ```

   Use this loop instruction to make it easier to explore the variations.

7. This exercise compares variables and cells. We give two code fragments. The first uses variables:

```
local X in
   X=23
   local X in
      X=44
   end
   {Browse X}
end
```

The second uses a cell:

```
local X in
   X={NewCell 23}
   X:=44
   {Browse @X}
end
```

In the first, the identifier `X` refers to two different variables. In the second, `X` refers to a cell. What does `Browse` display in each fragment? Explain.

8. This exercise investigates how to use cells together with functions. Let us define a function `{Accumulate N}` that accumulates all its inputs, i.e., it adds together all the arguments of all calls. Here is an example:

   ```
   {Browse {Accumulate 5}}
   {Browse {Accumulate 100}}
   {Browse {Accumulate 45}}
   ```

   This should display 5, 105, and 150, assuming that the accumulator contains zero at the start. Here is a wrong way to write `Accumulate`:

   ```
   declare
   fun {Accumulate N}
   Acc in
      Acc={NewCell 0}
      Acc:=@Acc+N
      @Acc
   end
   ```

   What is wrong with this definition? How would you correct it?

9. This exercise investigates another way of introducing state: a *memory store.* The memory store can be used to make an improved version of `FastPascal` that remembers previously-calculated rows.

   (a) A memory store is similar to the memory of a computer. It has a series of memory cells, numbered from 1 up to the maximum used so far. There are four functions on memory stores: `NewStore` creates a new store, `Put` puts a new value in a memory cell, `Get` gets the current value stored in a memory cell, and `Size` gives the highest-numbered cell used so far. For example:

```
declare
S={NewStore}
{Put S 2 [22 33]}
{Browse {Get S 2}}
{Browse {Size S}}
```

This stores `[22 33]` in memory cell 2, displays `[22 33]`, and then displays 2. Load into the Mozart system the memory store as defined in the supplements file on the book's Web site. Then use the interactive interface to understand how the store works.

(b) Now use the memory store to write an improved version of `FastPascal`, called `FasterPascal`, that remembers previously-calculated rows. If a call asks for one of these rows, then the function can return it directly without having to recalculate it. This technique is sometimes called *memoization* since the function makes a "memo" of its previous work. This improves its performance. Here's how it works:

- First make a store S available to `FasterPascal`.
- For the call `{FasterPascal N}`, let M be the number of rows stored in S, i.e., rows 1 up to M are in S.
- If N>M then compute rows M+1 up to N and store them in S.
- Return the Nth row by looking it up in S.

Viewed from the outside, `FasterPascal` behaves identically to `FastPascal` except that it is faster.

(c) We have given the memory store as a library. It turns out that the memory store can be defined by using a memory cell. We outline how it can be done and you can write the definitions. The cell holds the store contents as a list of the form `[N1|X1 ... Nn|Xn]`, where the cons `Ni|Xi` means that cell number `Ni` has content `Xi`. This means that memory stores, while they are convenient, do not introduce any additional expressive power over memory cells.

(d) Section 1.13 defines a counter with just one operation, `Bump`. This means that it is not possible to read the counter without adding one to it. This makes it awkward to use the counter. A practical counter would have at least two operations, say `Bump` and `Read`, where `Read` returns the current count without changing it. The practical counter looks like this:

```
declare
local C in
   C={NewCell 0}
   fun {Bump}
```

```
         C:=@C+1
         @C
      end
      fun {Read}
         @C
      end
   end
```

Change your implementation of the memory store so that it uses this counter to keep track of the store's size.

10. Section 1.15 gives an example using a cell to store a counter that is incremented by two threads.

    (a) Try executing this example several times. What results do you get? Do you ever get the result 1? Why could this be?

    (b) Modify the example by adding calls to `Delay` in each thread. This changes the thread interleaving without changing what calculations the thread does. Can you devise a scheme that always results in 1?

    (c) Section 1.16 gives a version of the counter that never gives the result 1. What happens if you use the delay technique to try to get a 1 anyway?

Part II

General Computation Models

Chapter 2

Declarative Computation Model

"Non sunt multiplicanda entia praeter necessitatem."
"*Do not multiply entities beyond necessity.*"
– Ockham's Razor, *William of Ockham* (1285–1349?)

Programming encompasses three things:

- First, a *computation model*, which is a formal system that defines a language and how sentences of the language (e.g., expressions and statements) are executed by an abstract machine. For this book, we are interested in computation models that are useful and intuitive for programmers. This will become clearer when we define the first one later in this chapter.
- Second, a set of programming techniques and design principles used to write programs in the language of the computation model. We will sometimes call this a *programming model*. A programming model is always built on top of a computation model.
- Third, a set of reasoning techniques to let you reason about programs, to increase confidence that they behave correctly and to calculate their efficiency.

The above definition of computation model is very general. Not all computation models defined in this way will be useful for programmers. What is a reasonable computation model? Intuitively, we will say that a reasonable model is one that can be used to solve many problems, that has straightforward and practical reasoning techniques, and that can be implemented efficiently. We will have more to say about this question later on. The first and simplest computation model we will study is *declarative programming*. For now, we define this as evaluating functions over partial data structures. This is sometimes called *stateless* programming, as opposed to *stateful* programming (also called *imperative* programming) which is explained in Chapter 6.

The declarative model of this chapter is one of the most fundamental computation models. It encompasses the core ideas of the two main declarative paradigms, namely functional and logic programming. It encompasses programming with functions over complete values, as in Scheme and Standard ML. It also encompasses deterministic logic programming, as in Prolog when search is not used. And finally, it can be made concurrent without losing its good properties (see Chapter 4).

Declarative programming is a rich area – most of the ideas of the more expressive computation models are already there, at least in embryonic form. We therefore present it in two chapters. This chapter defines the computation model and a practical language based on it. The next chapter, Chapter 3, gives the programming techniques of this language. Later chapters enrich the basic model with many concepts. Some of the most important are exception handling, concurrency, components (for programming in the large), capabilities (for encapsulation and security), and state (leading to objects and classes). In the context of concurrency, we will talk about dataflow, lazy execution, message passing, active objects, monitors, and transactions. We will also talk about user interface design, distribution (including fault tolerance), and constraints (including search).

Structure of the chapter

The chapter consists of seven sections:

- Section 2.1 explains how to define the syntax and semantics of practical programming languages. Syntax is defined by a context-free grammar extended with language constraints. Semantics is defined in two steps: by translating a practical language into a simple kernel language and then giving the semantics of the kernel language. These techniques will be used throughout the book. This chapter uses them to define the declarative computation model.

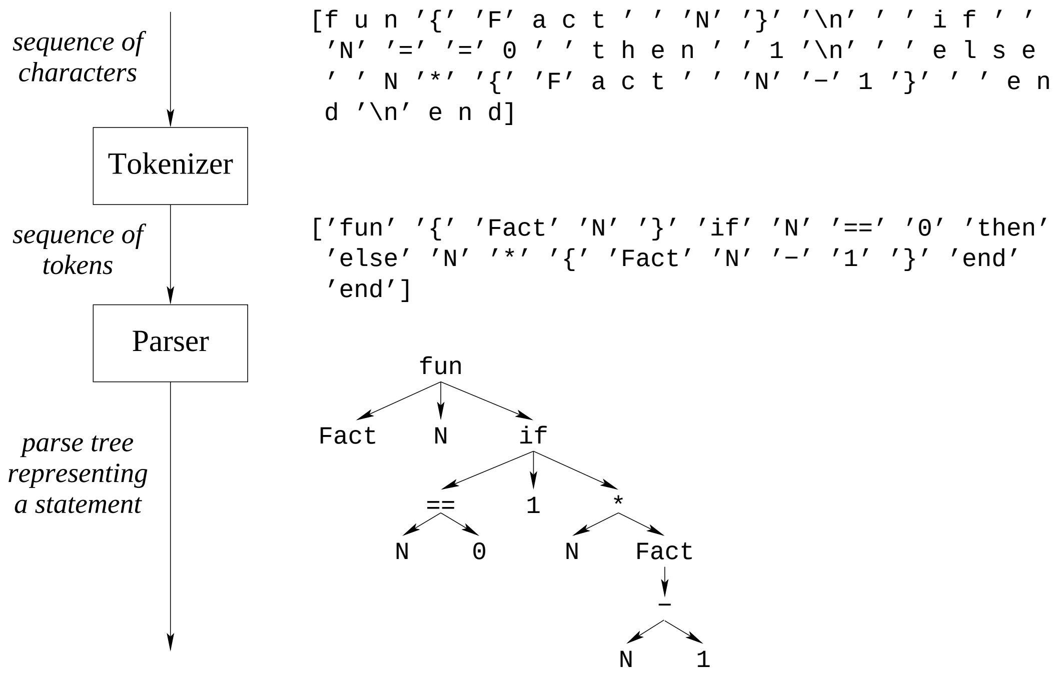
- The next three sections define the syntax and semantics of the declarative model:

  - Section 2.2 gives the data structures: the single-assignment store and its contents, partial values and dataflow variables.
  - Section 2.3 defines the kernel language syntax.
  - Section 2.4 defines the kernel language semantics in terms of a simple abstract machine. The semantics is designed to be intuitive and to permit straightforward reasoning about correctness and complexity.

- Section 2.5 defines a practical programming language on top of the kernel language.

- Section 2.6 extends the declarative model with *exception handling*, which allows programs to handle unpredictable and exceptional situations.

- Section 2.7 gives a few advanced topics to let interested readers deepen their understanding of the model.



```
[f u n '{' 'F' a c t ' ' 'N' '}' '\n' ' ' i f ' '
 'N' '=' '=' 0 ' ' t h e n ' ' 1 '\n' ' ' e l s e
 ' ' N '*' '{' 'F' a c t ' ' 'N' '-' 1 '}' ' ' e n
 d '\n' e n d]
```

```
['fun' '{' 'Fact' 'N' '}' 'if' 'N' '==' '0' 'then'
 'else' 'N' '*' '{' 'Fact' 'N' '-' '1' '}' 'end'
 'end']
```

Figure 2.1: From characters to statements

2.1 Defining practical programming languages

Programming languages are much simpler than natural languages, but they can still have a surprisingly rich syntax, set of abstractions, and libraries. This is especially true for languages that are used to solve real-world problems, which we call *practical* languages. A practical language is like the toolbox of an experienced mechanic: there are many different tools for many different purposes and all tools are there for a reason.

This section sets the stage for the rest of the book by explaining how we will present the syntax ("grammar") and semantics ("meaning") of practical programming languages. With this foundation we will be ready to present the first computation model of the book, namely the declarative computation model. We will continue to use these techniques throughout the book to define computation models.

2.1.1 Language syntax

The syntax of a language defines what are the *legal* programs, i.e., programs that can be successfully executed. At this stage we do not care what the programs are actually doing. That is semantics and will be handled in the next section.

Grammars

A grammar is a set of rules that defines how to make 'sentences' out of 'words'. Grammars can be used for natural languages, like English or Swedish, as well as for artificial languages, like programming languages. For programming languages, 'sentences' are usually called 'statements' and 'words' are usually called 'tokens'. Just as words are made of letters, tokens are made of characters. This gives us two levels of structure:

statement ('sentence')	=	sequence of tokens ('words')
token ('word')	=	sequence of characters ('letters')

Grammars are useful both for defining statements and tokens. Figure 2.1 gives an example to show how character input is transformed into a statement. The example in the figure is the definition of `Fact`:

```
fun {Fact N}
   if N==0 then 1
   else N*{Fact N-1} end
end
```

The input is a sequence of characters, where `´ ´` represents the space and `´\n´` represents the newline. This is first transformed into a sequence of tokens and subsequently into a parse tree. The syntax of both sequences in the figure is compatible with the list syntax we use throughout the book. Whereas the sequences are "flat", the parse tree shows the structure of the statement. A program that accepts a sequence of characters and returns a sequence of tokens is called a *tokenizer* or *lexical analyzer*. A program that accepts a sequence of tokens and returns a parse tree is called a *parser*.

Extended Backus-Naur Form

One of the most common notations for defining grammars is called Extended Backus-Naur Form (EBNF for short), after its inventors John Backus and Peter Naur. The EBNF notation distinguishes terminal symbols and nonterminal symbols. A *terminal symbol* is simply a token. A *nonterminal symbol* represents a sequence of tokens. The nonterminal is defined by means of a grammar rule, which shows how to expand it into tokens. For example, the following rule defines the nonterminal ⟨digit⟩:

$$\langle\textsf{digit}\rangle \quad ::= \quad 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

It says that ⟨digit⟩ represents one of the ten tokens 0, 1, ..., 9. The symbol "|" is read as "or"; it means to pick one of the alternatives. Grammar rules can themselves refer to other nonterminals. For example, we can define a nonterminal ⟨int⟩ that defines how to write positive integers:

$$\langle\textsf{int}\rangle \quad ::= \quad \langle\textsf{digit}\rangle\ \{\ \langle\textsf{digit}\rangle\ \}$$

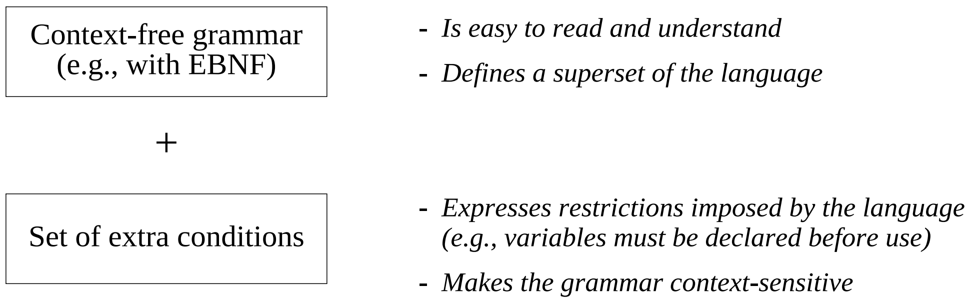

Context-free grammar
(e.g., with EBNF)
- Is easy to read and understand
- Defines a superset of the language
+
Set of extra conditions
- Expresses restrictions imposed by the language (e.g., variables must be declared before use)
- Makes the grammar context-sensitive

Figure 2.2: The context-free approach to language syntax

This rule says that an integer is a digit followed by zero or more digits. The braces "{ ... }" mean to repeat whatever is inside any number of times, including zero.

How to read grammars

To read a grammar, start with any nonterminal symbol, say ⟨int⟩. Reading the corresponding grammar rule from left to right gives a sequence of tokens according to the following scheme:

- Each terminal symbol encountered is added to the sequence.
- For each nonterminal symbol encountered, read its grammar rule and replace the nonterminal by the sequence of tokens that it expands into.
- Each time there is a choice (with |), pick any of the alternatives.

The grammar can be used both to verify that a statement is legal and to generate statements.

Context-free and context-sensitive grammars

Any well-defined set of statements is called a *formal language*, or *language* for short. For example, the set of all possible statements generated by a grammar and one nonterminal symbol is a language. Techniques to define grammars can be classified according to how expressive they are, i.e., what kinds of languages they can generate. For example, the EBNF notation given above defines a class of grammars called *context-free* grammars. They are so-called because the expansion of a nonterminal, e.g., ⟨digit⟩, is always the same no matter where it is used.

For most practical programming languages, there is usually no context-free grammar that generates all legal programs and no others. For example, in many languages a variable has to be declared before it is used. This condition cannot be expressed in a context-free grammar because the nonterminal that uses the

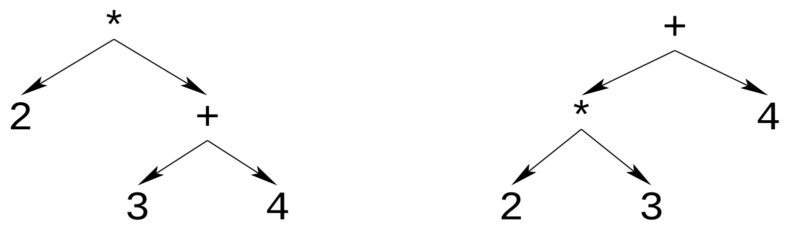


Figure 2.3: Ambiguity in a context-free grammar

variable must only allow using already-declared variables. This is a context dependency. A grammar that contains a nonterminal whose use depends on the context where it is used is called a *context-sensitive* grammar.

The syntax of most practical programming languages is therefore defined in two parts (see Figure 2.2): as a context-free grammar supplemented with a set of extra conditions imposed by the language. The context-free grammar is kept instead of some more expressive notation because it is easy to read and understand. It has an important locality property: a nonterminal symbol can be understood by examining only the rules needed to define it; the (possibly much more numerous) rules that use it can be ignored. The context-free grammar is corrected by imposing a set of extra conditions, like the declare-before-use restriction on variables. Taking these conditions into account gives a context-sensitive grammar.

Ambiguity

Context-free grammars can be *ambiguous*, i.e., there can be several parse trees that correspond to a given token sequence. For example, here is a simple grammar for arithmetic expressions with addition and multiplication:

$$\begin{array}{lcl} \langle\mathsf{exp}\rangle & ::= & \langle\mathsf{int}\rangle \mid \langle\mathsf{exp}\rangle\ \langle\mathsf{op}\rangle\ \langle\mathsf{exp}\rangle \\ \langle\mathsf{op}\rangle & ::= & + \mid * \end{array}$$

The expression `2*3+4` has two parse trees, depending on how the two occurrences of ⟨exp⟩ are read. Figure 2.3 shows the two trees. In one tree, the first ⟨exp⟩ is `2` and the second ⟨exp⟩ is `3+4`. In the other tree, they are `2*3` and `4`, respectively.

Ambiguity is usually an undesirable property of a grammar since it makes it unclear exactly what program is being written. In the expression `2*3+4`, the two parse trees give different results when evaluating the expression: one gives 14 (the result of computing 2*(3+4)) and the other gives 10 (the result of computing (2*3)+4). Sometimes the grammar rules can be rewritten to remove the ambiguity, but this can make the rules more complicated. A more convenient approach is to add extra conditions. These conditions restrict the parser so that only one parse tree is possible. We say that they *disambiguate* the grammar.

For expressions with binary operators such as the arithmetic expressions given above, the usual approach is to add two conditions, *precedence* and *associativity*:

- Precedence is a condition on an expression with different operators, like `2*3+4`. Each operator is given a *precedence level*. Operators with high precedences are put as deep in the parse tree as possible, i.e., as far away from the root as possible. If `*` has higher precedence than `+`, then the parse tree `(2*3)+4` is chosen over the alternative `2*(3+4)`. If `*` is deeper in the tree than `+`, then we say that `*` *binds tighter* than `+`.

- Associativity is a condition on an expression with the same operator, like `2-3-4`. In this case, precedence is not enough to disambiguate because all operators have the same precedence. We have to choose between the trees `(2-3)-4` and `2-(3-4)`. Associativity determines whether the leftmost or the rightmost operator binds tighter. If the associativity of `-` is `left`, then the tree `(2-3)-4` is chosen. If the associativity of `-` is `right`, then the other tree `2-(3-4)` is chosen.

Precedence and associativity are enough to disambiguate all expressions defined with operators. Appendix C gives the precedence and associativity of all the operators used in this book.

Syntax notation used in this book

In this chapter and the rest of the book, each new data type and language construct is introduced together with a small syntax diagram that shows how it fits in the whole language. The syntax diagram gives grammar rules for a simple context-free grammar of tokens. The notation is carefully designed to satisfy two basic principles:

- All grammar rules can stand on their own. No later information will ever invalidate a grammar rule. That is, we never give an incorrect grammar rule just to "simplify" the presentation.

- It is always clear by inspection when a grammar rule completely defines a nonterminal symbol or when it gives only a partial definition. A partial definition always ends in three dots "...".

All syntax diagrams used in the book are summarized in Appendix C. This appendix also gives the lexical syntax of tokens, i.e., the syntax of tokens in terms of characters. Here is an example of a syntax diagram with two grammar rules that illustrates our notation:

```
⟨statement⟩   ::=  skip | ⟨expression⟩ ´=´ ⟨expression⟩ | ...
⟨expression⟩  ::=  ⟨variable⟩ | ⟨int⟩ | ...
```

These rules give partial definitions of two nonterminals, ⟨statement⟩ and ⟨expression⟩. The first rule says that a statement can be the keyword **`skip`**, or two expressions separated by the equals symbol `=`, or something else. The second rule says that an expression can be a variable, an integer, or something else. To avoid confusion with the grammar rule's own syntax, a symbol that occurs literally in the text is always quoted with single quotes. For example, the equals symbol is shown as ´=´. Keywords are not quoted, since for them no confusion is possible. A choice between different possibilities in the grammar rule is given by a vertical bar |.

Here is a second example to give the remaining notation:

```
⟨statement⟩   ::=  if ⟨expression⟩ then ⟨statement⟩
                   { elseif ⟨expression⟩ then ⟨statement⟩ }
                   [ else ⟨statement⟩ ] end | ...
⟨expression⟩  ::=  ´[´ { ⟨expression⟩ }+ ´]´ | ...
⟨label⟩       ::=  unit | true | false | ⟨variable⟩ | ⟨atom⟩
```

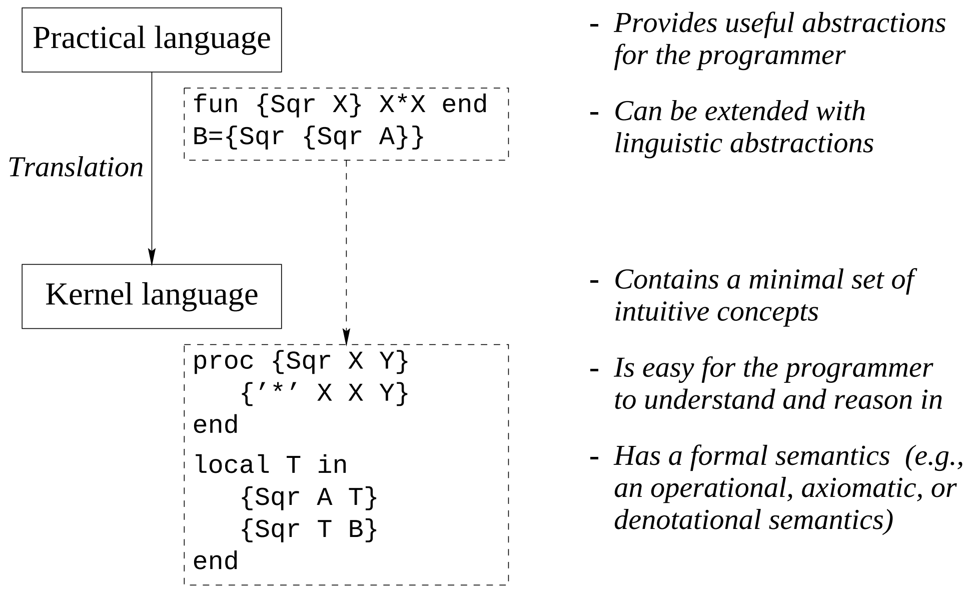
The first rule defines the **if** statement. There is an optional sequence of **elseif** clauses, i.e., there can be any number of occurrences including zero. This is denoted by the braces { ... }. This is followed by an optional **else** clause, i.e., it can occur zero or one times. This is denoted by the brackets [...]. The second rule defines the syntax of explicit lists. They must have at least one element, e.g., `[5 6 7]` is valid but `[ ]` is not (note the space that separates the `[` and the `]`). This is denoted by { ... }+. The third rule defines the syntax of record labels. This is a complete definition. There are five possibilities and no more will ever be given.

2.1.2 Language semantics

The semantics of a language defines what a program does when it executes. Ideally, the semantics should be defined in a simple mathematical structure that lets us reason about the program (including its correctness, execution time, and memory use) without introducing any irrelevant details. Can we achieve this for a practical language without making the semantics too complicated? The technique we use, which we call the kernel language approach, gives an affirmative answer to this question.

Modern programming languages have evolved through more than five decades of experience in constructing programmed solutions to complex, real-world problems.[1] Modern programs can be quite complex, reaching sizes measured in millions of lines of code, written by large teams of human programmers over many years. In our view, languages that scale to this level of complexity are successful in part because they model some essential aspects of how to construct complex programs. In this sense, these languages are not just arbitrary constructions of the human mind. We would therefore like to understand them in a scientific way, i.e., by explaining their behavior in terms of a simple underlying model. This is the deep motivation behind the kernel language approach.

[1]The figure of five decades is somewhat arbitrary. We measure it from the first working stored-program computer, the Manchester Mark I. According to lab documents, it ran its first program on June 21, 1948 [178].


Practical language
Translation
Kernel language
- Provides useful abstractions for the programmer
- Can be extended with linguistic abstractions
- Contains a minimal set of intuitive concepts
- Is easy for the programmer to understand and reason in
- Has a formal semantics (e.g., an operational, axiomatic, or denotational semantics)

```
fun {Sqr X} X*X end
B={Sqr {Sqr A}}
```

```
proc {Sqr X Y}
   {'*' X X Y}
end
local T in
   {Sqr A T}
   {Sqr T B}
end
```

Figure 2.4: The kernel language approach to semantics

The kernel language approach

This book uses the kernel language approach to define the semantics of programming languages. In this approach, all language constructs are defined in terms of translations into a core language known as the kernel language. The kernel language approach consists of two parts (see Figure 2.4):

- First, define a very simple language, called the *kernel language*. This language should be easy to reason in and be faithful to the space and time efficiency of the implementation. The kernel language and the data structures it manipulates together form the *kernel computation model*.
- Second, define a translation scheme from the full programming language to the kernel language. Each grammatical construct in the full language is translated into the kernel language. The translation should be as simple as possible. There are two kinds of translation, namely *linguistic abstraction* and *syntactic sugar*. Both are explained below.

The kernel language approach is used throughout the book. Each computation model has its kernel language, which builds on its predecessor by adding one new concept. The first kernel language, which is presented in this chapter, is called the *declarative kernel language*. Many other kernel languages are presented later on in the book.

Formal semantics

The kernel language approach lets us define the semantics of the kernel language in any way we want. There are four widely-used approaches to language semantics:

- An *operational semantics* shows how a statement executes in terms of an abstract machine. This approach always works well, since at the end of the day all languages execute on a computer.

- An *axiomatic semantics* defines a statement's semantics as the relation between the input state (the situation before executing the statement) and the output state (the situation after executing the statement). This relation is given as a logical assertion. This is a good way to reason about statement sequences, since the output assertion of each statement is the input assertion of the next. It therefore works well with stateful models, since a state is a sequence of values. Section 6.6 gives an axiomatic semantics of Chapter 6's stateful model.

- A *denotational semantics* defines a statement as a function over an abstract domain. This works well for declarative models, but can be applied to other models as well. It gets complicated when applied to concurrent languages. Sections 2.7.1 and 4.9.2 explain functional programming, which is particularly close to denotational semantics.

- A *logical semantics* defines a statement as a model of a logical theory. This works well for declarative and relational computation models, but is hard to apply to other models. Section 9.3 gives a logical semantics of the declarative and relational computation models.

Much of the theory underlying these different semantics is of interest primarily to mathematicians, not to programmers. It is outside the scope of the book to give this theory. The principal formal semantics we give in this book is an operational semantics. We define it for each computation model. It is detailed enough to be useful for reasoning about correctness and complexity yet abstract enough to avoid irrelevant clutter. Chapter 13 collects all these operational semantics into a single formalism with a compact and readable notation.

Throughout the book, we give an informal semantics for every new language construct and we often reason informally about programs. These informal presentations are always based on the operational semantics.

Linguistic abstraction

Both programming languages and natural languages can evolve to meet their needs. When using a programming language, at some point we may feel the need to extend the language, i.e., to add a new linguistic construct. For example, the declarative model of this chapter has no looping constructs. Section 3.6.3 defines a **`for`** construct to express certain kinds of loops that are useful for writing declarative programs. The new construct is both an abstraction and an addition to the language syntax. We therefore call it a *linguistic abstraction.* A practical programming language consists of a set of linguistic abstractions.

There are two phases to defining a linguistic abstraction. First, define a new grammatical construct. Second, define its translation into the kernel language. The kernel language is not changed. This book gives many examples of useful linguistic abstractions, e.g., functions (**fun**), loops (**for**), lazy functions (**fun lazy**), classes (**class**), reentrant locks (**lock**), and others.[2] Some of these are part of the Mozart system. The others can be added to Mozart with the `gump` parser-generator tool [104]. Using this tool is beyond the scope of this book.

A simple example of a linguistic abstraction is the function syntax, which uses the keyword **fun**. This is explained in Section 2.5.2. We have already programmed with functions in Chapter 1. But the declarative kernel language of this chapter only has procedure syntax. Procedure syntax is chosen for the kernel since all arguments are explicit and there can be multiple outputs. There are other, deeper reasons for choosing procedure syntax which are explained later in this chapter. Because function syntax is so useful, though, we add it as a linguistic abstraction.

We define a syntax for both function definitions and function calls, and a translation into procedure definitions and procedure calls. The translation lets us answer all questions about function calls. For example, what does `{F1 {F2 X} {F3 Y}}` mean exactly (nested function calls)? Is the order of these function calls defined? If so, what is the order? There are many possibilities. Some languages leave the order of argument evaluation unspecified, but assume that a function's arguments are evaluated before the function. Other languages assume that an argument is evaluated when and if its result is needed, not before. So even as simple a thing as nested function calls does not necessarily have an obvious semantics. The translation makes it clear what the semantics is.

Linguistic abstractions are useful for more than just increasing the expressiveness of a program. They can also improve other properties such as correctness, security, and efficiency. By hiding the abstraction's implementation from the programmer, the linguistic support makes it impossible to use the abstraction in the wrong way. The compiler can use this information to give more efficient code.

Syntactic sugar

It is often convenient to provide a short-cut notation for frequently-occurring idioms. This notation is part of the language syntax and is defined by grammar rules. This notation is called *syntactic sugar*. Syntactic sugar is analogous to linguistic abstraction in that its meaning is defined precisely by translating it into the full language. But it should not be confused with linguistic abstraction: it does not provide a new abstraction, but just reduces program size and improves program readability.

We give an example of syntactic sugar that is based on the **local** statement.

[2]Logic gates (**gate**) for circuit descriptions, mailboxes (**receive**) for message-passing concurrency, and currying and list comprehensions as in modern functional languages, cf., Haskell.

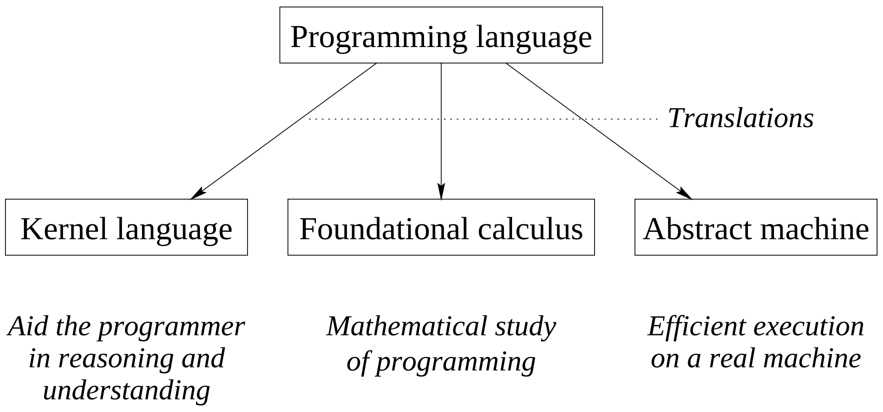

Programming language
Translations
Kernel language
Foundational calculus
Abstract machine
Aid the programmer in reasoning and understanding
Mathematical study of programming
Efficient execution on a real machine

Figure 2.5: Translation approaches to language semantics

Local variables can always be defined by using the statement **local** X **in ... end**. When this statement is used inside another, it is convenient to have syntactic sugar that lets us leave out the keywords **local** and **end**. Instead of:

```
if N==1 then [1]
else
   local L in
      ...
   end
end
```

we can write:

```
if N==1 then [1]
else L in
   ...
end
```

which is both shorter and more readable than the full notation. Other examples of syntactic sugar are given in Section 2.5.1.

Language design

Linguistic abstractions are a basic tool for language design. Any abstraction that we define has three phases in its lifecycle. When first we define it, it has no linguistic support, i.e., there is no syntax in the language designed to make it easy to use. If at some point, we suspect that it is especially basic and useful, we can decide to give it linguistic support. It then becomes a linguistic abstraction. This is an exploratory phase, i.e., there is no commitment that the linguistic abstraction will become part of the language. If the linguistic abstraction is successful, i.e., it simplifies programs and is useful to programmers, then it becomes part of the language.

Other translation approaches

The kernel language approach is an example of a *translation approach* to semantics, i.e., it is based on a translation from one language to another. Figure 2.5 shows the three ways that the translation approach has been used for defining programming languages:

- The kernel language approach, used throughout the book, is intended for the programmer. Its concepts correspond directly to programming concepts.
- The foundational approach is intended for the mathematician. Examples are the Turing machine, the λ calculus (underlying functional programming), first-order logic (underlying logic programming), and the π calculus (to model concurrency). Because these calculi are intended for formal mathematical study, they have as few elements as possible.
- The machine approach is intended for the implementor. Programs are translated into an idealized machine, which is traditionally called an *abstract machine* or a *virtual machine*.[3] It is relatively easy to translate idealized machine code into real machine code.

Because we focus on practical programming techniques, this book uses only the kernel language approach.

The interpreter approach

An alternative to the translation approach is the interpreter approach. The language semantics is defined by giving an interpreter for the language. New language features are defined by extending the interpreter. An *interpreter* is a program written in language L_1 that accepts programs written in another language L_2 and executes them. This approach is used by Abelson & Sussman [2]. In their case, the interpreter is *metacircular*, i.e., L_1 and L_2 are the same language L. Adding new language features, e.g., for concurrency and lazy evaluation, gives a new language L' which is implemented by extending the interpreter for L.

The interpreter approach has the advantage that it shows a self-contained implementation of the linguistic abstractions. We do not use the interpreter approach in this book because it does not in general preserve the execution-time complexity of programs (the number of operations needed as a function of input size). A second difficulty is that the basic concepts interact with each other in the interpreter, which makes them harder to understand.

[3]Strictly speaking, a virtual machine is a software emulation of a real machine, running on the real machine, that is almost as efficient as the real machine. It achieves this efficiency by executing most virtual instructions directly as real instructions. The concept was pioneered by IBM in the early 1960's in the VM operating system. Because of the success of Java, which uses the term "virtual machine", modern usage tends to blur the distinction between abstract and virtual machines.

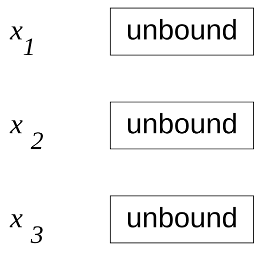


Figure 2.6: A single-assignment store with three unbound variables

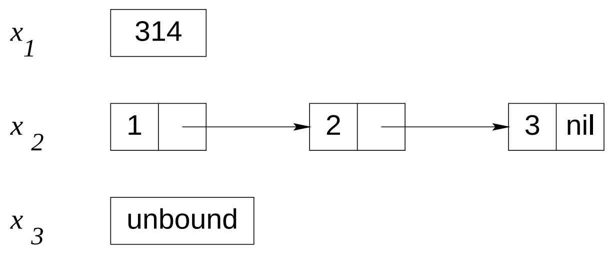


Figure 2.7: Two of the variables are bound to values

2.2 The single-assignment store

We introduce the declarative model by first explaining its data structures. The model uses a *single-assignment store*, which is a set of variables that are initially unbound and that can be bound to one value. Figure 2.6 shows a store with three unbound variables x_1, x_2, and x_3. We can write this store as $\{x_1, x_2, x_3\}$. For now, let us assume we can use integers, lists, and records as values. Figure 2.7 shows the store where x_1 is bound to the integer 314 and x_2 is bound to the list `[1 2 3]`. We write this as $\{x_1 = 314, x_2 = \texttt{[1 2 3]}, x_3\}$.

2.2.1 Declarative variables

Variables in the single-assignment store are called *declarative* variables. We use this term whenever there is a possible confusion with other kinds of variables. Later on in the book, we will also call these variables *dataflow* variables because of their role in dataflow execution.

Once bound, a declarative variable stays bound throughout the computation and is indistinguishable from its value. What this means is that it can be used in calculations as if it were the value. Doing the operation $x + y$ is the same as doing $11 + 22$, if the store is $\{x = 11, y = 22\}$.

2.2.2 Value store

A store where all variables are bound to values is called a *value store.* Another way to say this is that a value store is a persistent mapping from variables to

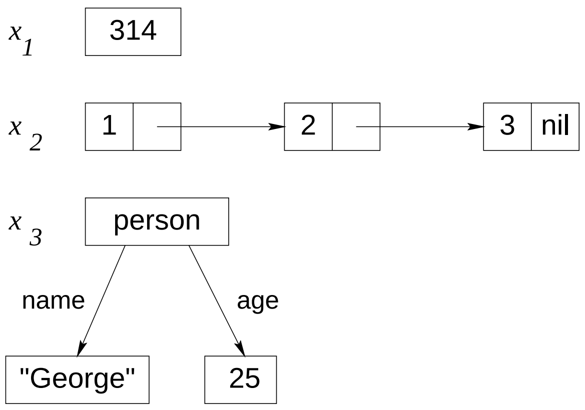


Figure 2.8: A value store: all variables are bound to values

values. A *value* is a mathematical constant. For example, the integer 314 is a value. Values can also be compound entities. For example, the list `[1 2 3]` and the record `person(name:"George" age:25)` are values. Figure 2.8 shows a value store where x_1 is bound to the integer 314, x_2 is bound to the list `[1 2 3]`, and x_3 is bound to the record `person(name:"George" age:25)`. Functional languages such as Standard ML, Haskell, and Scheme get by with a value store since they compute functions on values. (Object-oriented languages such as Smalltalk, C++, and Java need a cell store, which consists of cells whose content can be modified.)

At this point, a reader with some programming experience may wonder why we are introducing a single-assignment store, when other languages get by with a value store or a cell store. There are many reasons. The first reason is that we want to compute with partial values. For example, a procedure can return an output by binding an unbound variable argument. The second reason is declarative concurrency, which is the subject of Chapter 4. It is possible because of the single-assignment store. The third reason is that it is essential when we extend the model to deal with relational (logic) programming and constraint programming. Other reasons having to do with efficiency (e.g., tail recursion and difference lists) will become clear in the next chapter.

2.2.3 Value creation

The basic operation on a store is binding a variable to a newly-created value. We will write this as x_i=*value*. Here x_i refers directly to a variable in the store (and is *not* the variable's textual name in a program!) and *value* refers to a value, e.g., 314 or `[1 2 3]`. For example, Figure 2.7 shows the store of Figure 2.6 after the two bindings:

$x_1 = 314$
$x_2 =$ `[1 2 3]`

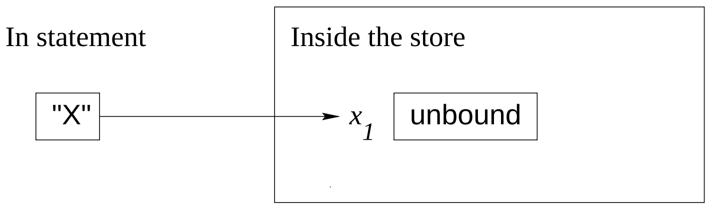


Figure 2.9: A variable identifier referring to an unbound variable

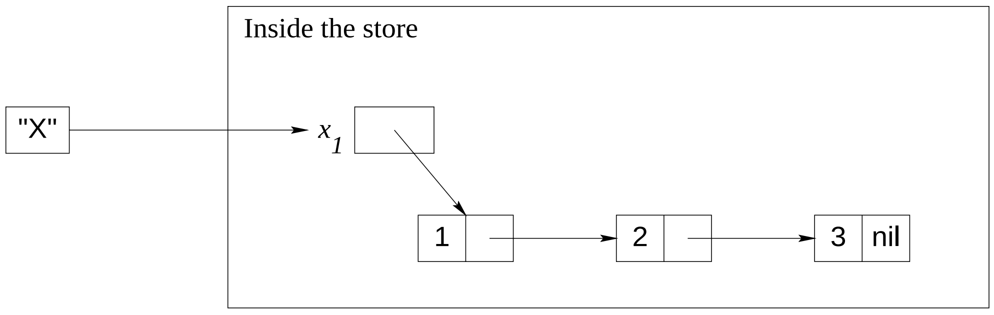


Figure 2.10: A variable identifier referring to a bound variable

The single-assignment operation x_i=*value* constructs *value* in the store and then binds the variable x_i to this value. If the variable is already bound, the operation will test whether the two values are compatible. If they are not compatible, an error is signaled (using the exception-handling mechanism, see Section 2.6).

2.2.4 Variable identifiers

So far, we have looked at a store that contains variables and values, i.e., *store entities*, with which calculations can be done. It would be nice if we could refer to a store entity from outside the store. This is the role of variable identifiers. A *variable identifier* is a textual name that refers to a store entity from outside the store. The mapping from variable identifiers to store entities is called an *environment*.

The variable names in program source code are in fact variable identifiers. For example, Figure 2.9 has an identifier "X" (the capital letter X) that refers to the store variable x_1. This corresponds to the environment $\{\mathsf{X} \rightarrow x_1\}$. To talk about *any* identifier, we will use the notation $\langle\mathsf{x}\rangle$. The environment $\{\langle\mathsf{x}\rangle \rightarrow x_1\}$ is the same as before, if $\langle\mathsf{x}\rangle$ represents X. As we will see later, variable identifiers and their corresponding store entities are added to the environment by the **local** and **declare** statements.

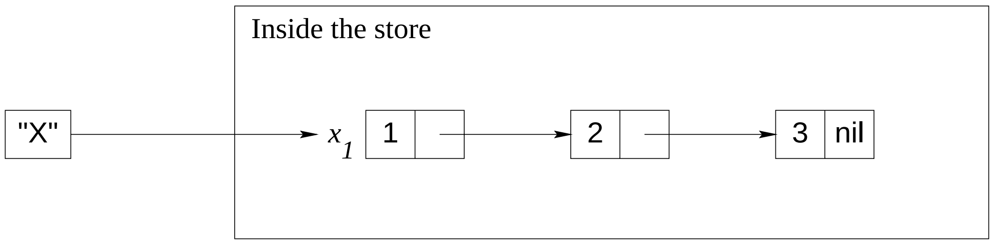


Figure 2.11: A variable identifier referring to a value

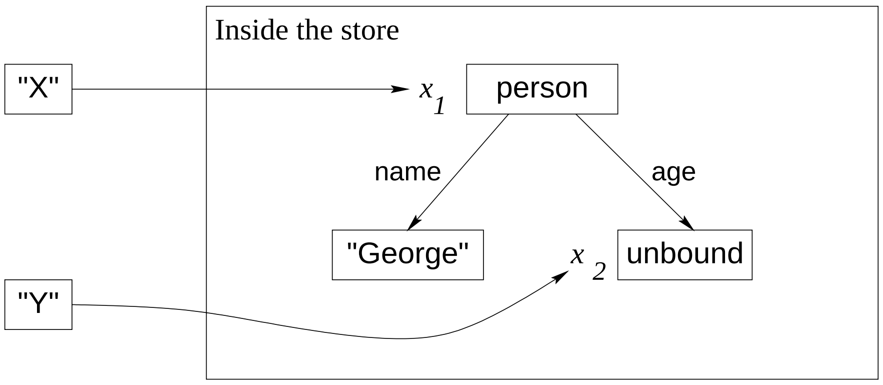


Figure 2.12: A partial value

2.2.5 Value creation with identifiers

Once bound, a variable is indistinguishable from its value. Figure 2.10 shows what happens when x_1 is bound to `[1 2 3]` in Figure 2.9. With the variable identifier `X`, we can write the binding as `X=[1 2 3]`. This is the text a programmer would write to express the binding. We can also use the notation ⟨x⟩`=[1 2 3]` if we want to be able to talk about *any* identifier. To make this notation legal in a program, ⟨x⟩ has to be replaced by an identifier.

The equality sign "`=`" refers to the bind operation. After the bind completes, the identifier "`X`" still refers to x_1, which is now bound to `[1 2 3]`. This is indistinguishable from Figure 2.11, where `X` refers directly to `[1 2 3]`. Following the links of bound variables to get the value is called *dereferencing*. It is invisible to the programmer.

2.2.6 Partial values

A *partial value* is a data structure that may contain unbound variables. Figure 2.12 shows the record `person(name:"George" age:`x_2`)`, referred to by the identifier `X`. This is a partial value because it contains the unbound variable x_2. The identifier `Y` refers to x_2. Figure 2.13 shows the situation after x_2 is bound to 25 (through the bind operation `Y=25`). Now x_1 is a partial value with no unbound variables, which we call a *complete value*. A declarative variable can